TerraWeather
Ensuring Your Safety & Success

AI Workshop – Lecture 5

Time-series Data

5.0 Time-Series Data

What differentiates time-series from other types of data is of course, *time*. This means that data points have a specific relation to each other, and this must be exploited if you wish to create a successful predictive model.

This aspect of time and relation of data through time, needs special attention when you train your NNs. You need to:

- Avoid simple point-wise inputs that don't capture the aspect of time.
- Avoid high dimensional input data from small samples. This is known as the “curse of dimensionality”.
- Avoid simple point-wise loss functions like Euclidean loss that don't take into account the aspect of time.

This lecture will unpack the first two of these ideas. The last idea will be taken up in Lecture 6.

5.1 Simple Inputs

The NNs you've constructed so far are *stateless*. This means that if you apply an input you'll get the same answer. Once trained, a stateless NN behaves like a function $f(\boldsymbol{x})$.

The statelessness of your NNs means that they have no notion of time. For example, consider the time-ordered data/label training pairs (x,y):

`(0,4) (0,4) (0,4) (0,400) (1,5) (1,5) (1,5) (1,500) (2,6) (2,6)(2,6)(2,600)`

A stateless NN can never learn the obvious pattern in time for the labels. The training label/prediction graph may show underpredicted peaks, or an inability to learn the labels at all.

The reason is because we have trained the NNs with input that don't capture the aspect of time.

5.2 Explicit Time

One method is to include time explicitly.

This should **not** be done directly, for example $x_k \rightarrow [k, x_k]$ (**Quiz 5A.1:** why

not?) nor should you use a cyclic measure of time like the day of the year. (**Quiz 5A.2:** why not?)

Instead, you have to convert a cyclic measure of time – like day of year, week number or month number, depending on your input data – into a 2-component vector representing time.

For example, if you have daily data, you could use the Julian Day *t* as an explicit time measure. You then include into your data the sine and cosine of *t* divided by its cycle (in this case, 365):

$$\begin{aligned} x_t &\rightarrow [\sin(z), \cos(z), x_t] \\ z &= \frac{2\pi t}{365} \end{aligned}$$ – Equation 5.0

Quiz 5A:

1. Explain clearly why we shouldn't perform the change $x_k \rightarrow [k, x_k]$ where k is an absolute time measure (eg, time in milliseconds from a reference date)

2. Explain clearly why we shouldn't perform the change $x_t \rightarrow [t, x_t]$, where t is a cyclic time measure (eg, day of year).

3. * Explain why we need both sine and cosine terms in Equation 5.0.

4. Kasman wants to predict the shopping behaviour of customers in his mall. He wants to predict customer by day of week. What explicit time input should he use?

5.3 Windowing

Simple techniques like explicit time can be very useful for highly cyclic data (eg, load forecasting), but is less helpful when the data shows no obvious cyclic variation with time.

Continuing our earlier example, another method might be to re-group every 2 inputs, forming a higher dimensional input vector:

```
([0,0], 4) ([0,0], 4) ([0,1], 400) ([0,1], 5) ([1,1], 5) ([1,1], 500) ...
```

Here we have used a *sliding window* of length 2 to capture the time change in

the input. Essentially, by artificially increasing the input dimension, we form a "fingerprint" that the NN can learn:

$$\begin{aligned} x_0 &= 0 \rightarrow (discarded) \\ x_1 &= 0 \rightarrow [0,0] \\ x_2 &= 0 \rightarrow [0,0] \\ x_3 &= 1 \rightarrow [0,1] \\ x_4 &= 1 \rightarrow [1,1] \\ x_5 &= 1 \rightarrow [1,1] \\ x_6 &= 2 \rightarrow [1,2] \\ x_7 &= 2 \rightarrow [2,2] \\ x_8 &= 2 \rightarrow [2,2] \end{aligned}$$

Windowing input data has some problems:

1. We had to discard the first training pair. In general, a window of size N will discard N-1 training pairs.

2. The training pairs are still not entirely disambiguated. Note that the input vector [1,1] is mapped to both 5 and 500.

3. The longer vector will encourage *memorization* (ie, the network essentially encodes the training data in its weights) and therefore poor generalization. This is very likely for small time-series. This problem is related to the increase in dimensions, and not windowing specifically.

4. For realistic time-series, the inputs become highly correlated, and this can impede learning. This is likely for longer time-series.

Nevertheless, the idea of a sliding window producing a fingerprint is very useful in time-series prediction. In upcoming lectures, we see how to lessen the problems caused by windowing.

We will discuss the problems (3) and (4) next.

5.4 The Curse of High Dimensions

The problem with artificially increasing the input dimensions to create a high dimensional fingerprint is that the number of data-points (which is constant, since it will be roughly the size of our training set) per unit volume of hyperspace (of dimension *d*) decreases exponentially as 2^{1-d} . This means

that we will greatly undersample the input space we are interested in modeling. To see this clearly, in the example of section 5.1:

- The input dimension *d* is 1, and the input space is the subspace $[0,2]$ of $\mathbb{R}^1$. So, it has a volume 2 – 0 = 2 units.
- There are 12 data points in the training dataset, so the density is 12/2 = 6 points per unit volume.

When we expand the input dimension from 1 to 2 by windowing,

- The input dimension *d* is 2, and the input space is the subspace $[0,2]\times[0,2]$ of $\mathbb{R}^2$. So, it has a volume 2x2 = 4 units.
- So, the data density is 12 / 4 = 3 points per unit volume.

So, we have *halved* the data point density just by increasing the input dimension by 1.

In realistic situations, to adequately capture time and get a good fingerprint, we would be forced to increase the input dimension to the 100s, so this will cause the resulting high dimensional input space to be woefully undersampled.

Neural networks are very prone to memorization (which is the cause of overfit) when the density of points is very small.

This is a challenging situation, but can be solved in two ways – by manually culling unnecessary dimensions or by automatically mapping the higher dimensional space into a much smaller one. We tackle the latter in depth in Lectures 7 & 8 using *stacked autoencoders*.

Quiz 5B

1. * In the windowing example, the data points fall within a region of $[0,2]\times[0,2]$ Plot the points and determine the shape of this region. Can the dimension of this subspace be compressed while keeping the benefits of the fingerprint? How could dimensional reduction work in practice?

2. * How would you go about manually culling “unnecessary” dimensions?

3. What are some possible drawbacks of manually culling “unnecessary” dimensions? What might be some advantages?

4. * Can you convince yourself why NNs memorize when the training sample density is very small? Hint: you can use the ideas in Lecture 3.

5. ** Suggest a possible mechanism by which NNs memorize their training data.

6. ** Research some other approaches apart from NNs that don't really suffer from the Curse of Dimensionality. (What are their drawbacks though?)

5.5 Highly Correlated Inputs

The last problem that can slow down learning or cause it to fail is the presence of highly correlated inputs. Take a situation where the *j*-th input is the same as the *k*-th input. In this case, you can imagine that eliminating the weights to one or the other in any input-layer neuron would not affect the end result of training.

In practice, the BP algorithm wastes time looking for intermediate combinations and may get stuck with sub-optimal training, or (more likely), be tempted to memorize due to the larger input dimension.

So, it is important to avoid selecting inputs that are highly correlated.

5.6 Input Normalization

Consider the situation where two inputs Z_1 and Z_2 into a certain neuron α are always positive. This is often the case for many time-series data. At each iteration of the BP algorithm, the weights are updated by a term proportional to:

$$\epsilon_{\alpha+1}\frac{\partial X_\alpha}{\partial W_j}=\epsilon_{\alpha+1}\frac{\partial \sum W_j Z_j+b_\alpha}{\partial W_j}=\epsilon_{\alpha+1}Z_j$$

Note that according to this equation, both Z_1 and Z_2 must increase or decrease together in the same direction as $sign(\epsilon_{\alpha+1})$. If the final targeted values of W_1 and W_2 are very different, this implies that their intermediate values must needlessly oscillate up and down (since they have to move in tandem) to reach their desired values. This phenomena both slows down learning or possibly causes it to fail. This problem remains as long as the means of the variables are non-zero.

So, we need to subtract from each training (and test!) time-series input, the training input average:

$$z'_k(t) = \frac{z_k(t) - \mu_k}{\sigma_k}$$ – Equation 5.6

Note that in a multi-layer feedforward network, this analysis can be applied to each successive layer also – the means of outputs of lower layers should be comparable since they are the "inputs" of layers above them. So, mean subtraction is useful even for internal nodes.

In Equation 5.6, we also scale the time-series by their variances. This has shown to improve convergence.

5.7 Hidden Variables

In the transformative processes involved from input to output – are there possible hidden variables?

For example, are temperature and rainfall data sufficient to predict dengue outbreaks? The answer is likely "no" because at best, they predict ideal mosquito breeding conditions. These conditions need to be correlated with population data in order to predict an outbreak. So, one hidden variable would be population data.

A useful proxy for hidden variables is the label itself. In the dengue prediction case, if *d(t+16)* were the label then it would be wise to include *d(t)* as an input. This helps your NN compensate for possible hidden variables like population. The drawback is that without careful handling, your predictions could be drawn towards the persistence benchmark.

TerraWeather
Ensuring Your Safety & Success

Lab 5: TW extensions to Caffe

In this lab, you will

1. Try two normalization schemes
2. Use the average layer to reduce dependency between inputs

One advantage of using Caffe is its easy extensibility. You have already encountered a modification to standard Caffe developed by Terra Weather to implement early stopping. Besides changing the solver behaviour, new layers can also be made to manipulate the blobs in the exact way you want. We will introduce three non-standard Caffe layers that performs data normalization and windowing will be useful in almost all time-series prediction problems.

`Norm` layer

This layer normalises its input blob by subtracting the mean and dividing by the standard deviation.

$$x'_k = \frac{x_k - \mu}{\sigma}$$

An example of how to declare the layer in a prototext file is shown here

```
layer {
  name: "norm"
  type: "Norm"

  bottom: "data"
  top: "norm_data"

  norm_param {
    axis: 2
    split: true
    length: 100
    once: true
  }
}
```

There are 4 main parameters that define the normalization operation. Only the first `length` number of time-steps (first axis of blob) are used to calculate the mean and standard deviation, ie

$$\mu = \frac{1}{\text{length}} \sum_{k=1}^{\text{length}} x_k \quad \text{and} \quad \sigma = \frac{1}{\text{length}} \sum_{k=1}^{\text{length}} (x_k - \mu)^2$$

This is useful in the test phases, where normalization over the training data only is desired so that the calculated scaling constants will be the same in both train and test phases.

It is sometimes inappropriate to normalize all data together, for example if there are several input variables in the same blob then it makes sense to normalize each variable separately. The axes which are normalized separately starts from the second layer (`axis=1`) up to the axis number specified by the `axis` parameter. Whether the axis at the `axis` parameter itself is normalized together depends on the `split` parameter (`true`: separate, `false`: together). In the above example, `axis=0` and `3` onwards (if they exist) of the data blob will be normalized together while `axis=2` is treated as separate variables.

The `once` parameter is meant for use with small datasets like those used in this dengue prediction problem. If the dataset is small enough to fit entirely in memory, then the input (HDF5Data) layer batch size can be set equal to the dataset size, such that the entire dataset is loaded in a single iteration. In this case, the input layer is just reloading the entire dataset into its output blob at every iteration. Since the input data is always the same, the normalization can be done just once to avoid unnecessary computation. This is the default behaviour and can be disabled by setting the `once` parameter to `false`.

There is an optional `scale` parameter that multiplies the output blob by a scalar multiplier after the normalization operations.

This layer is normally used right after the input layer for both data and label blobs. Two norm layers are needed for that as only one bottom blob is allowed per layer.

`InputScale` layer
This layer provides an alternative normalization method. It scales its input to fit within two values specified by the `min` and `max` parameters. This layer also has the `length`, `axis` and `split` parameters that have the same purpose as in the Norm layer. The once parameter however is set to true and cannot be changed. Hence this layer can only be used for small datasets that fit wholly in memory.

`Average` layer
The average layer calculates the moving average along the first axis (`axis=0`) within a number of time-steps determined by the `size` parameter. The `once` parameter is, similar to the above two layer types, used for small datasets and defaults to `true`.

Instructions
Use the dengue dataset provided in Lab 4 for your experiments.

Using the normalization layers
Both normalization layers determine their scaling constants over the training set which is assumed to be the first `length` values of the input. This requires the following changes to the set up in previous labs

- `test.txt`: the training dataset has to be fed as input for the scaling constants to be calculated
- Input layer batch size: the batch sizes of the input layer has to match the size of the training dataset during the training phase and the combined train and test dataset in the testing phase.
- Additional `Slice` layer: only the predictions for the test dataset are wanted during the test phase. This requires a `Slice` layer (and `Silence` layer) to remove the train dataset after normalization.

Compare and contrast the two types of normalization:

- Is one normalization method always better than the other? Which is more suitable for each input variable used in the dengue prediction problem?
- Set up experiments to show any effect of switching between the two normalization methods. Are the results consistent with your expectation?

Reducing input variable dependence

- Compare the use of a size 20 window as input (input size of 20 * 3 values per time step) versus the moving average of the same variables (3 values).
- What averaging size will you use? Are they the same for all variables? How would you implement that?
- Would having 2 averaging windows of size 10 instead of a single size 20 window give an improvement? What about larger number of smaller windows?